



Efficient Memory Management

[Vincenzo Innocente - CERN](#)

[Original lectures by Giulio Eulisse - CERN and Lassi Tuura \(FNAL, now Google\)](#)

About These Lectures

These lectures will address memory use and management in large scale scientific computing applications, with Linux/C++ focus.

I will introduce general concepts mainly through specific concrete examples common to everyday developer work. I will focus on common aspects on commodity hardware, in areas I am personally experienced in – this is not a tour of absolutely everything there is to know about memory management.

<http://inf-nesc.github.io/esc18/memory>
All the exercise material for these lectures

Additional Reading

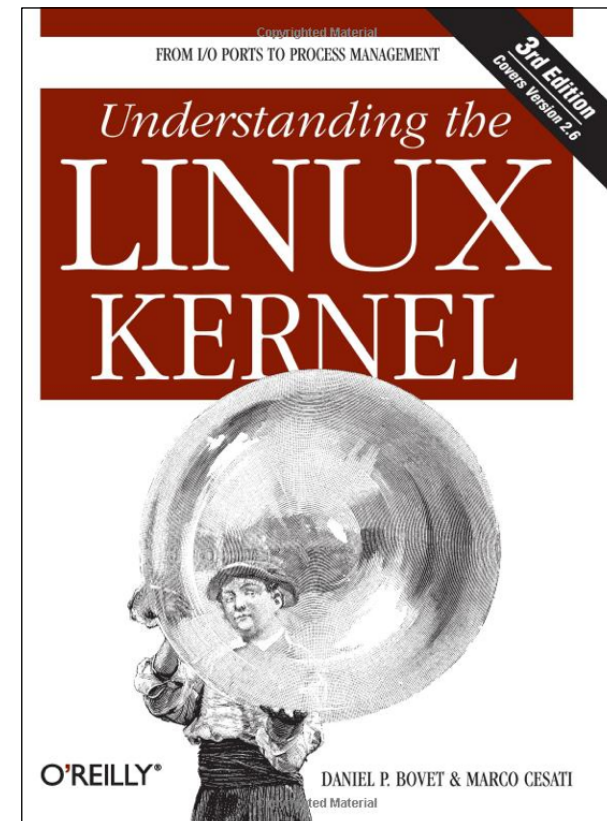
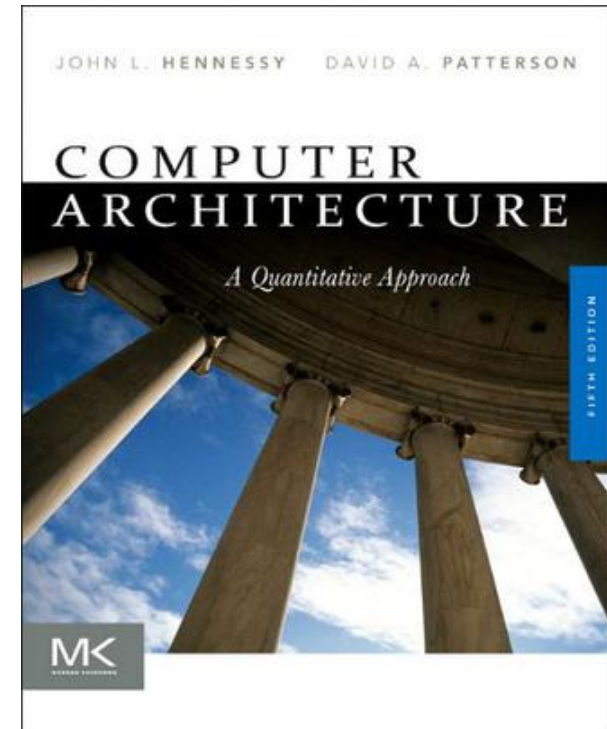
J. Hennessy, D. Patterson,
Computer Architecture: A Quantitative Approach,
5th edition (2011), ISBN 978-0-12-383872-8

U. Drepper,
What Every Programmer Should Know About Memory,
<http://people.redhat.com/drepper/cpumemory.pdf>

D. Bovet, M. Cesati,
Understanding the Linux Kernel,
3rd Edition, O'Reilly 2005, ISBN 0-596-00565-2

<http://techreport.com>, reviews with technical detail

<http://jemalloc.net> one of the best memory manager



Why Memory Management Matters?

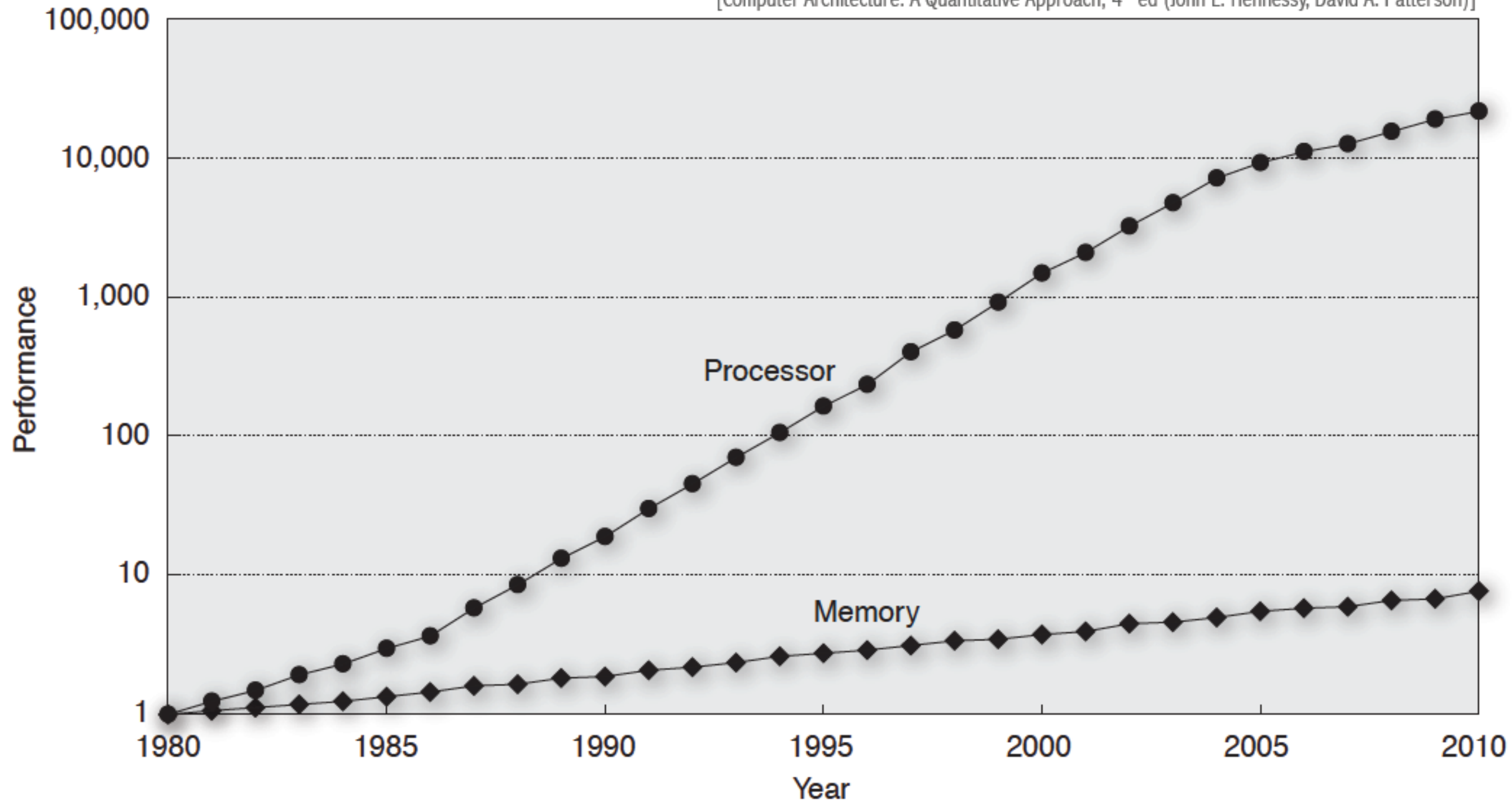
So, you've got a problem to solve. You've designed an algorithm to solve it. Now all you need is to code it up and you are done, right?

Actually, you have just begun. Your algorithm will translate to *real machine code*, which will run on very *real physical systems*, which have very *real practical limitations*.

A complete design must account for the real world limitations. This means “*the solution*” *will vary over time* with technology evolution.

The Performance Gap

[Computer Architecture: A Quantitative Approach, 4th ed (John L. Hennessy, David A. Patterson)]



Memory performance evolution compared with processor performance

Why Memory Management Matters?

Different solutions to the same problem vary dramatically in real life performance.

Algorithmic and data structure changes can easily result in several orders of magnitude improvement and regression. Always research this option first.

In some cases, changes in memory use and management can also easily produce orders of magnitude performance wins and losses – even without major logical change to the underlying algorithms. Common critical factors include memory churn, poor locality, and in multi-processing, memory contention.

In other cases, simple, subtle changes can yield performance wins in the 1-10% range. When % of your computing capacity is counted in rows of racks and days of processing, this still matters a great deal in practice! The small stuff still directly affects how much science you get out of your funding.

Memory Management at 10'000ft

Physical hardware

CPU pipelines and out-of-order execution; memory management unit [MMU] and physical memory banks and access properties; interconnect – front-side bus [FSB] vs. direct path [AMD: HT, Intel: QPI]; cache coherence and atomic operations; memory access non-uniformity [NUMA].

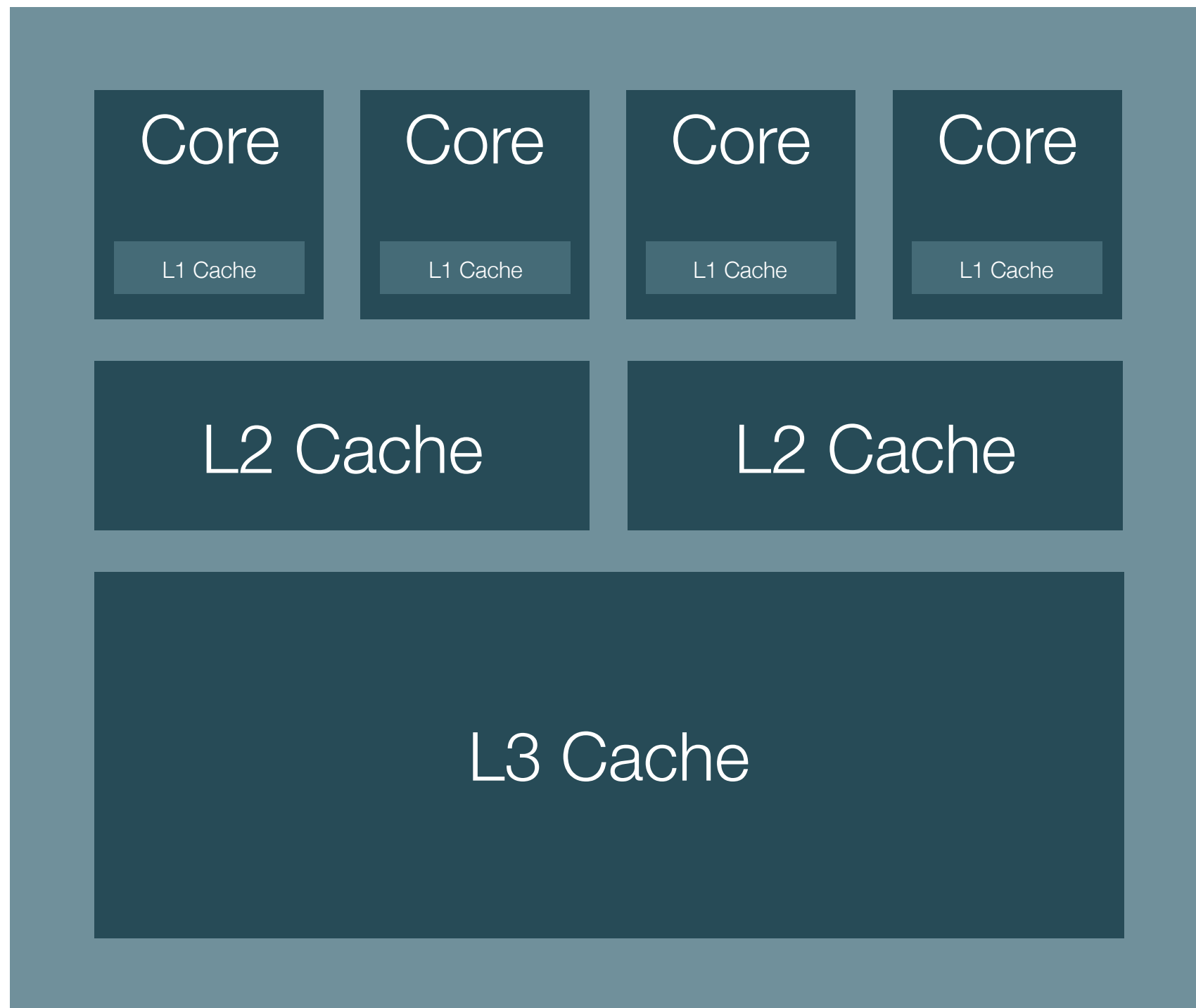
Operating system kernel

Per-process linear virtual address space; virtual memory translation from logical pages to physical page frames; page allocation and swapping; file and other caching; shared memory.

Run time

Code, data, heap, thread stacks; acquiring memory [sbrk/mmap]; sharing memory [shmget/mmap/fork]; C/C++ libraries and containers; application memory management.

Memory hierarchy

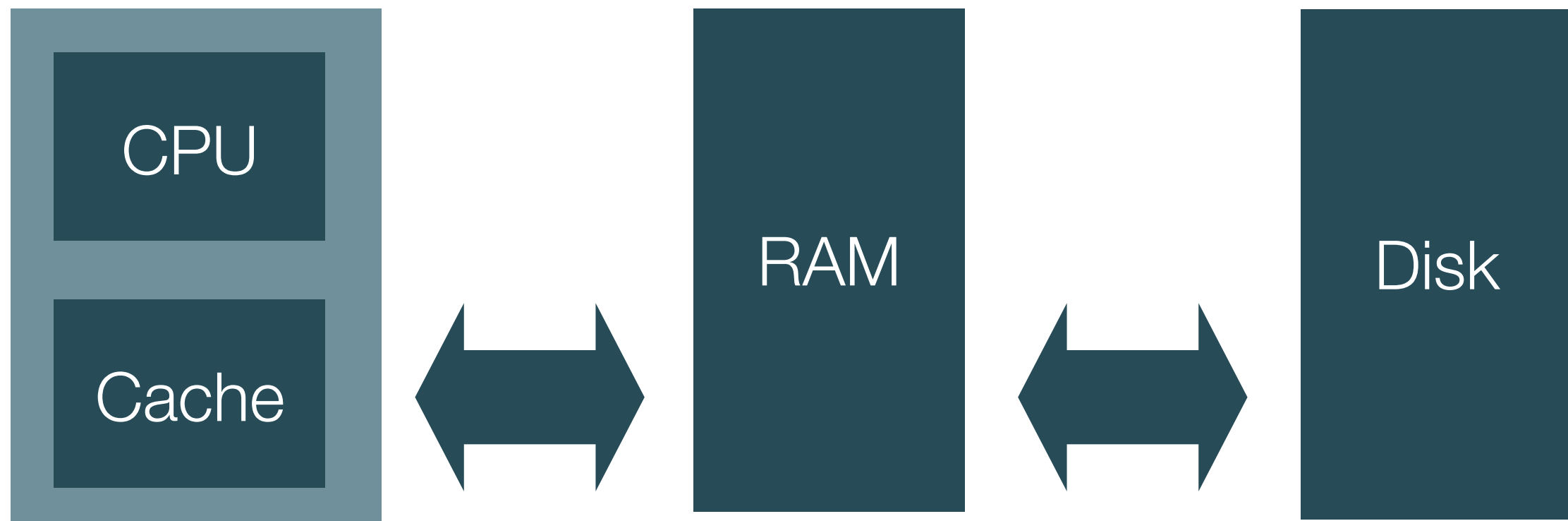


2 - 8 **cores** per die, 1 - 2 dies per package, 1-N packages per system.

3 levels of **cache**

- Small [32kB] separate L1 I+D caches for each core.
- Medium [256kB - 6MB] combined L2 cache, perhaps shared among some cores.
- Large [4 - 20MB] combined L3 cache shared between all cores on die.
- Can have even more exotic setups, especially when on cpu GPU is present.

Memory hierarchy



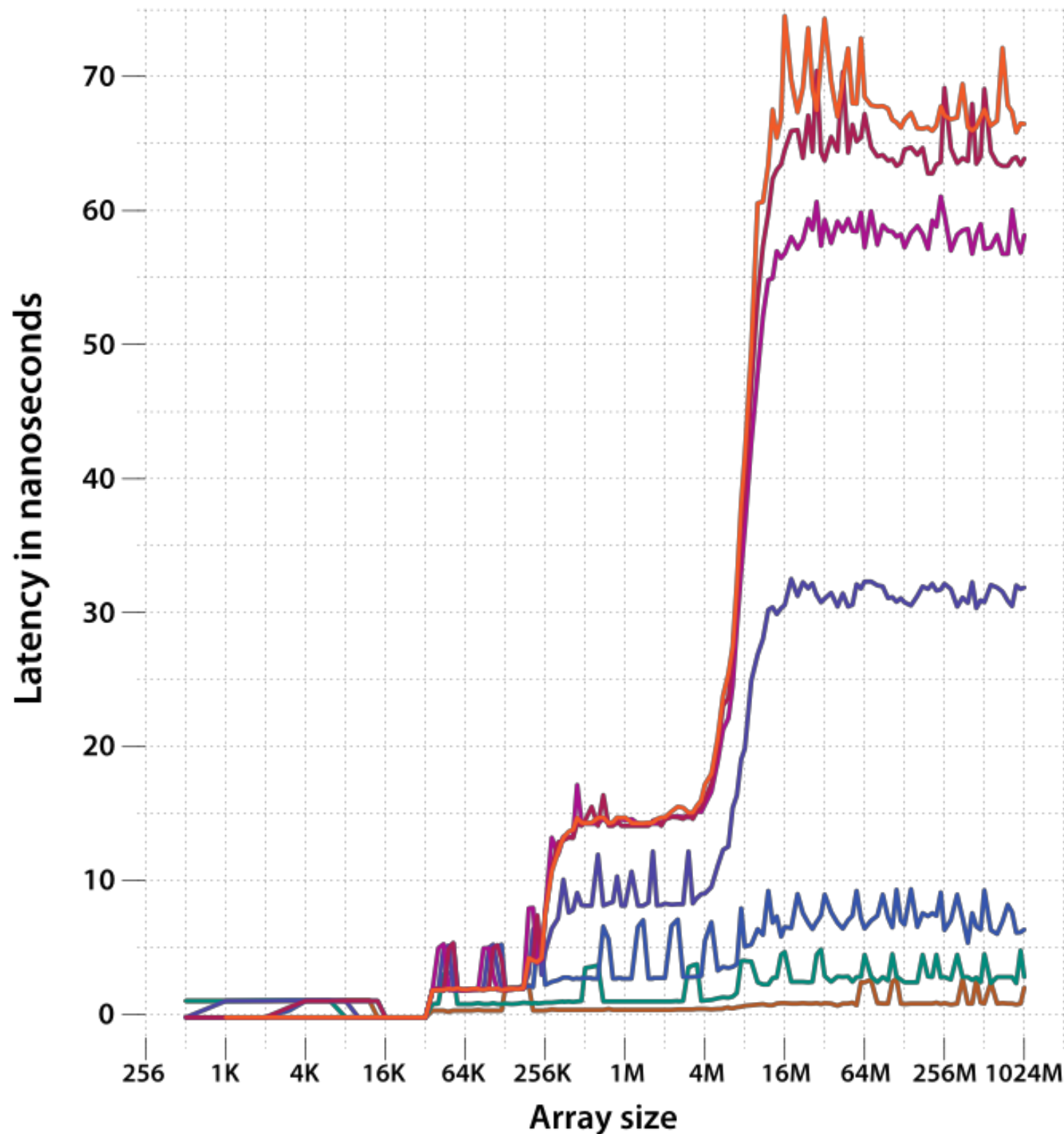
Exchange
cache-lines:
64 bytes*, aligned.

*Exchange **pages: 4096**
bytes, aligned.***

*: on most architectures

** : larger pages are available under certain cases

Memory latency, Linux 2.6.28 x86-64
Intel i7 940 2.93 GHz, 6GB



[LMBENCH 2.5 results for array strides 16, 32, 64, 256, 512, 1024B]

The Memory Wall

Average memory access time
$$= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty.}$$

I/D\$: L1 hit = 2-3 clock cycles.

I/D\$: L1 miss, L2 hit = ~ 10-15 cycles.

TLB: L1 miss, L2 hit = ~ 8-10 cycles.

TLB: L1 miss, L2 miss = ~ 30+ cycles.

What happens when you drop to memory?

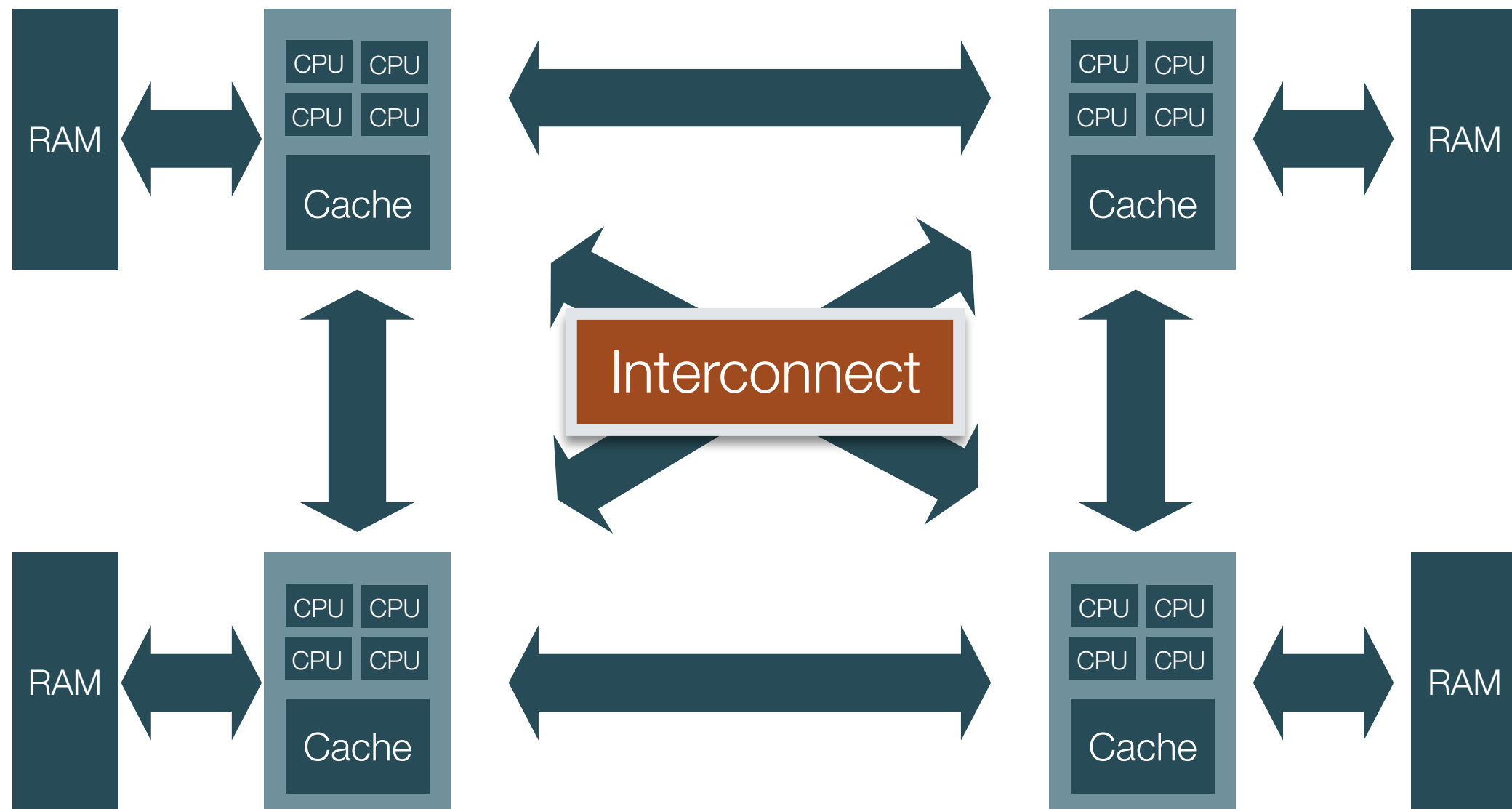
Intel Netburst Xeon (Pentium-era) memory latency was 400-700 clock cycles depending on access pattern and architecture.

AMD Opteron, Intel Core 2 and later CPU memory latency is ~200 cycles (times any NUMA overhead if crossing interconnect).

Good cache efficiency matters.

Non-Uniform memory access

RAM is not necessarily local anymore



Operating System and Memory

The operating system manages processes and their **address spaces**.

Each process has a virtual linear address space to itself, isolated from other address spaces and the kernel itself. Each process has **one or more threads**, which share the address space but have a separate stack and execution state.

The operating system manages **memory allocation and sharing**.

Memory is used for kernel itself and files in the **buffer cache**. Applications can share memory by referring to shared physical pages: just memory blocks, buffer cache regions, or special objects such as pipe memory with `vmsplice()`. Methods to share memory include **`fork()`**, **`mmap()`** or **`shmget()`**.

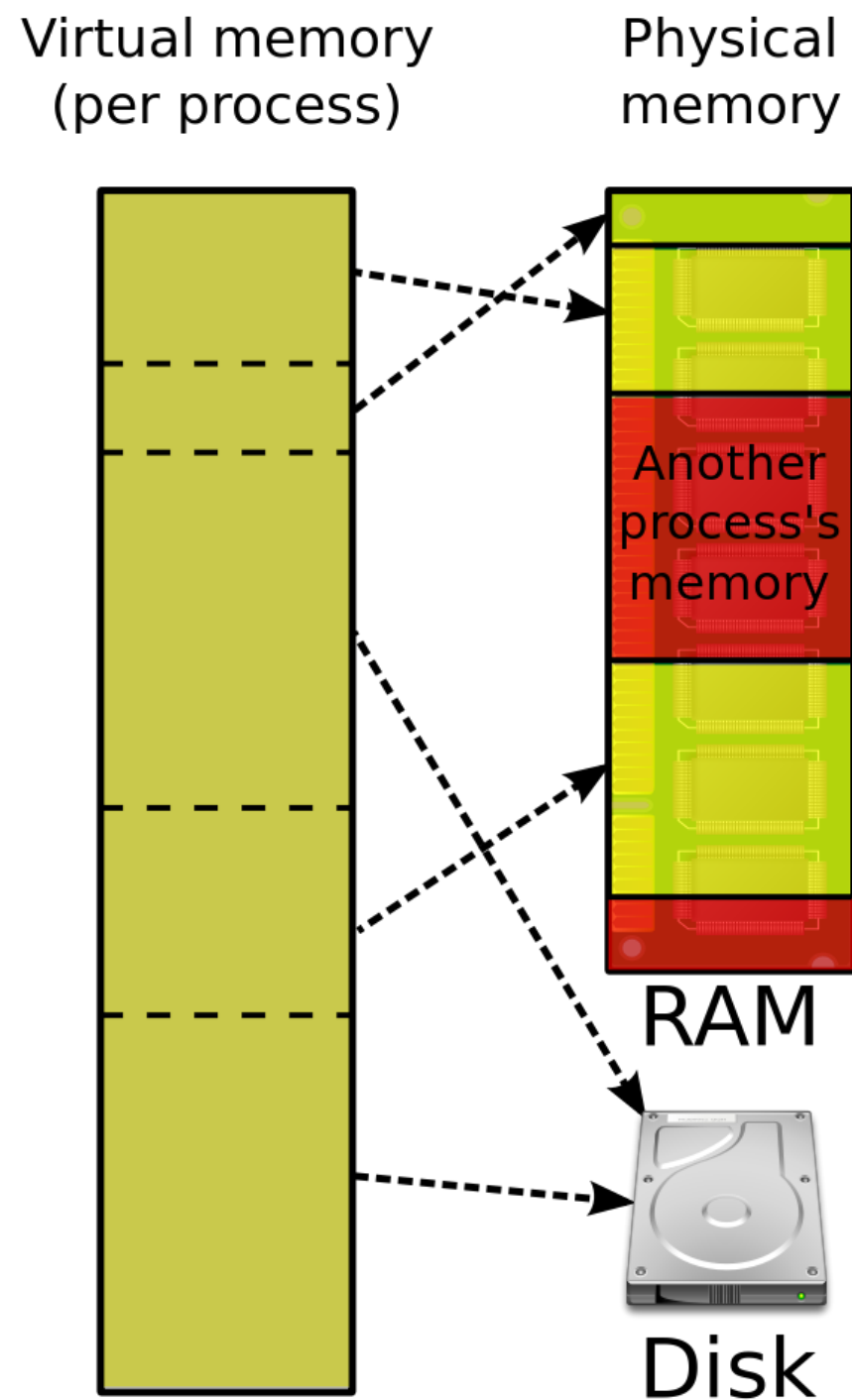
On **NUMA** systems the OS also manages process-to-physical memory mapping. In practice **application affinity hinting** is necessary (cf. `numactl`).

<http://man7.org/linux/man-pages/man1/top.1.html#OVERVIEW>

<http://man7.org/linux/man-pages/man5/proc.5.html>

<https://www.howtoforge.com/linux-pmap-command/>

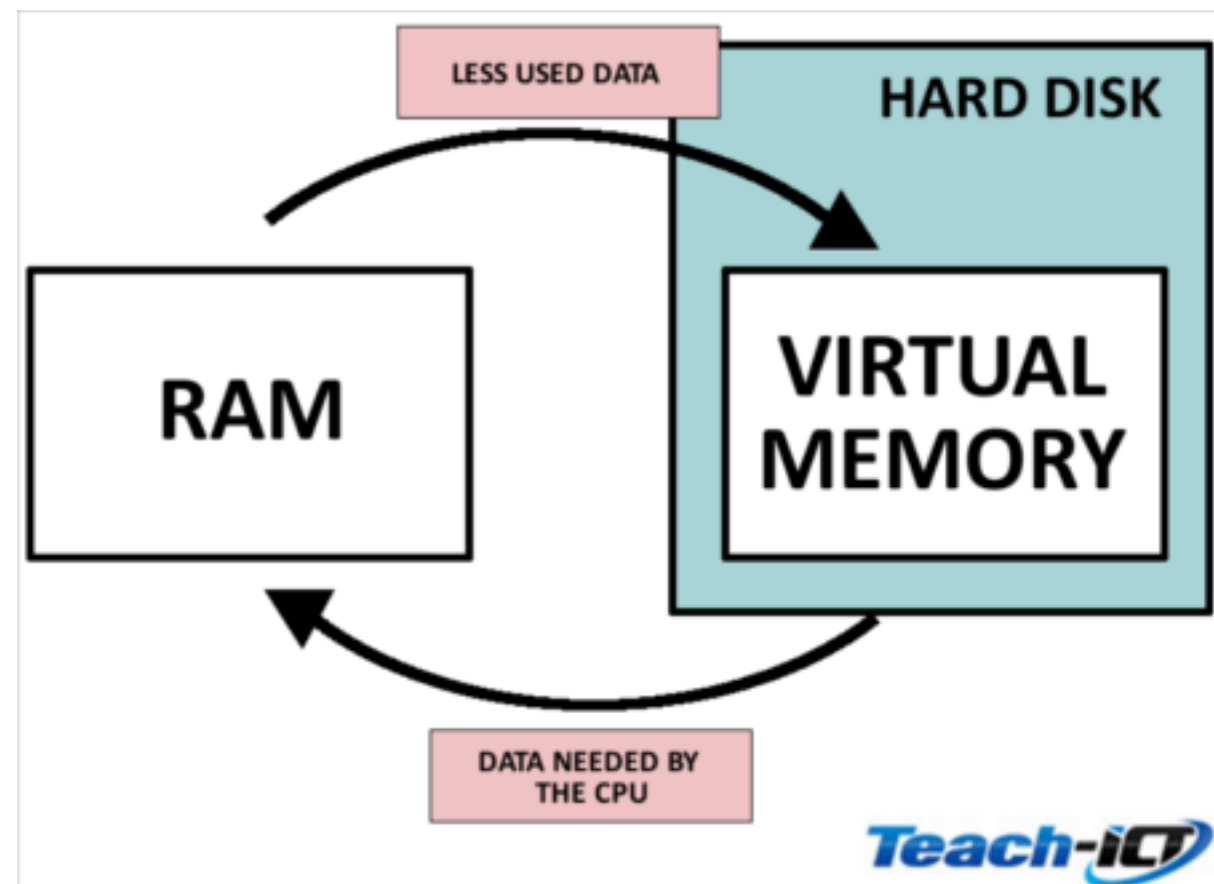
Virtual Memory



VSS

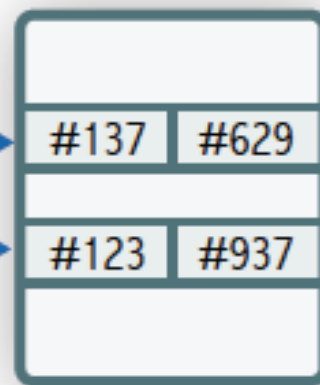
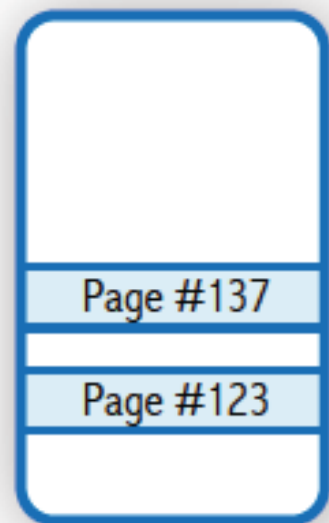
RSS

SWAP



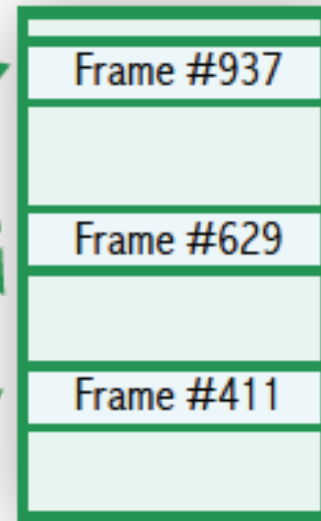
Virtual Memory

Process B Virtual
Address Space



OS Kernel
Page Tables

Process A Virtual
Address Space



Physical
Memory

Today's OSES give processes a flat* **linear virtual address space**: the same linear address in two different address spaces means two entirely different physical addresses.

Virtual and real physical memory is divided in **pages**, usually 4kB, but optionally 1-4MB. The OS provides the CPU per-process **page tables** to map a virtual address to a contiguous **physical page frame** plus offset, which in turn translates to memory bank, row and column.

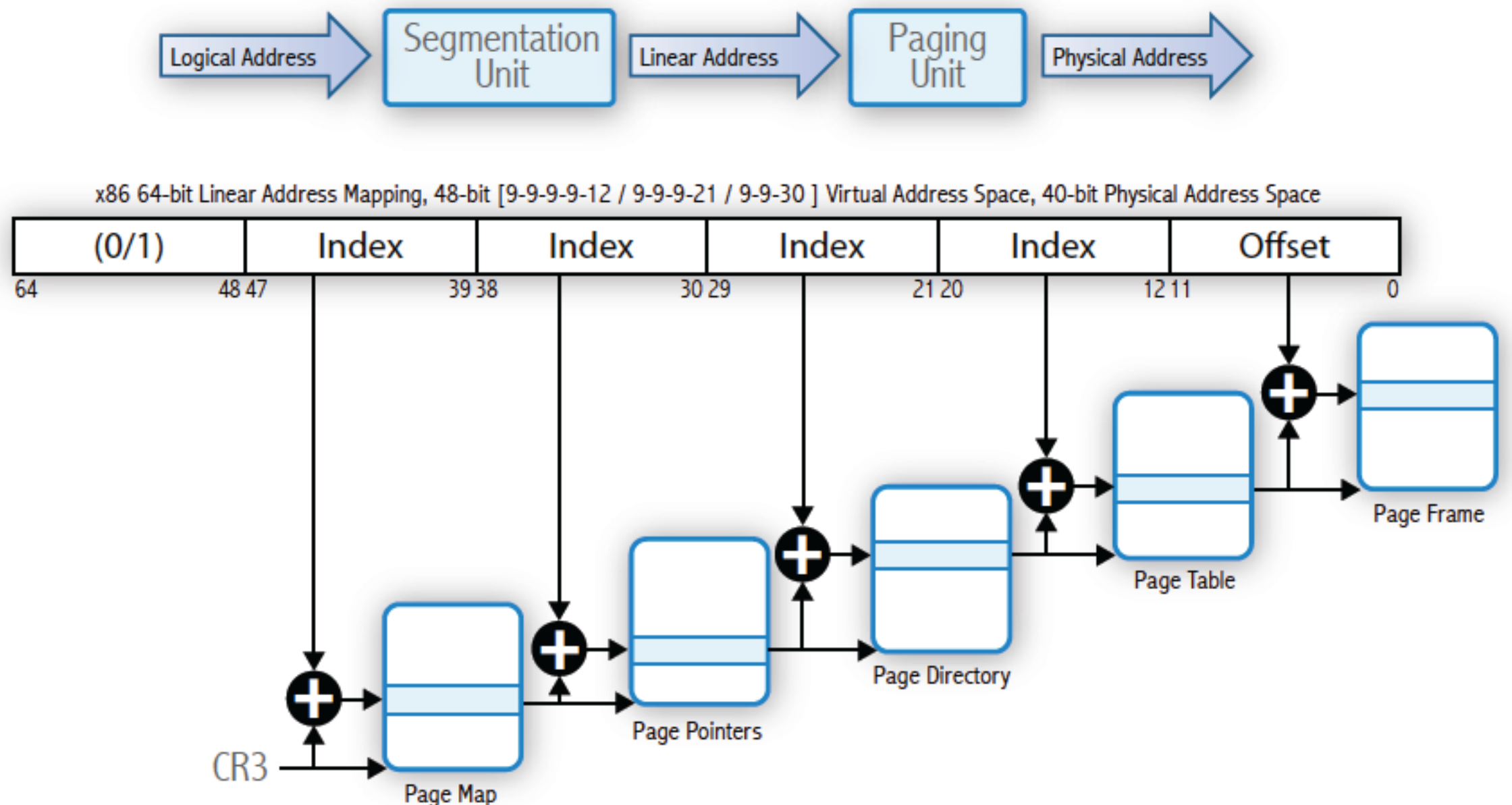
Page tables themselves **use memory, consume L2+ cache space**, and are never swapped out.

Even if processes share physical page frames, the **page tables are not shared**. With 4kB pages, large address spaces mean *big page tables*, even if the memory itself is shared: there's over 2MB of page tables for every 1GB of committed address space.+

* CPUs also segment or otherwise divide memory in regions; details in the references. "Flat" does not mean "simple", the address space can be a fairly hairy object.

+ 2GB VSIZE × 128 processes requires 0.5GB page tables.

Virtual Address Translation

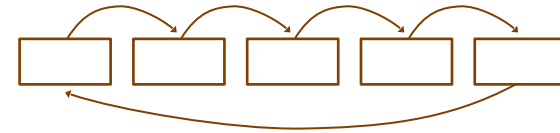


Special cache hardware called **TLB**, translation look-aside buffer, accelerates virtual-to-physical address mapping to avoid a full page table walk on every memory op. TLB fits only a **limited number of pages**.

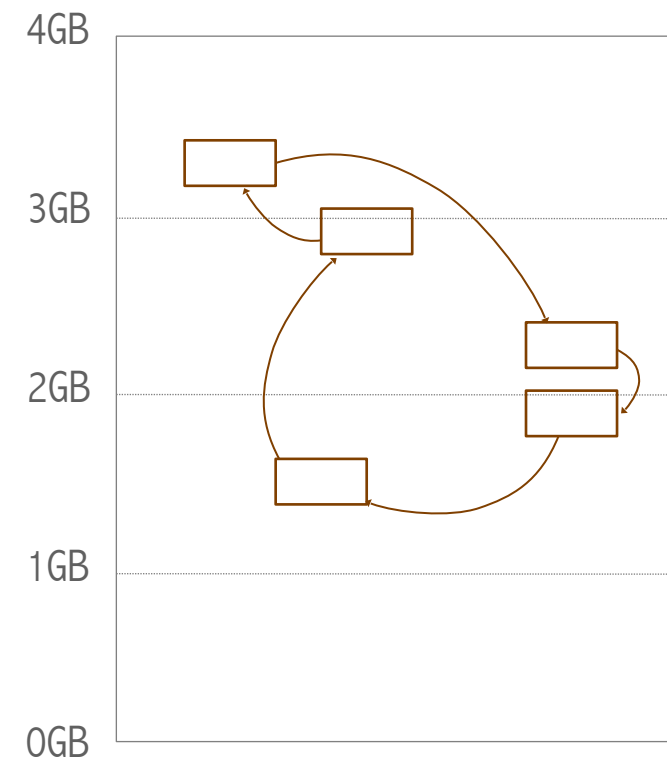
A page which isn't present or valid causes a **page fault**. The OS handles these, e.g. code page is read in from a file on disk on first use. Some page table changes force a synchronous update on all processors ("TLB shutdown").

Logical vs. Real Data Structures

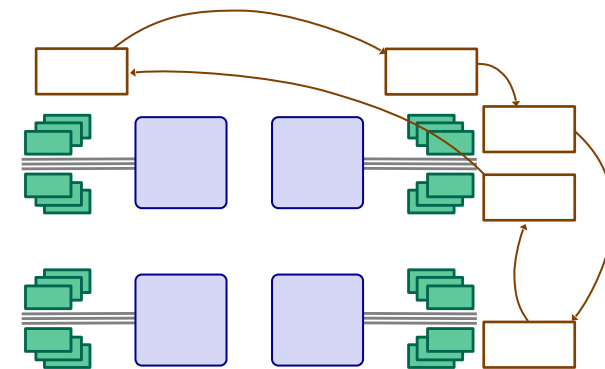
This logical linked list...



Could be scattered in virtual address space like this...



And in physical memory like this...

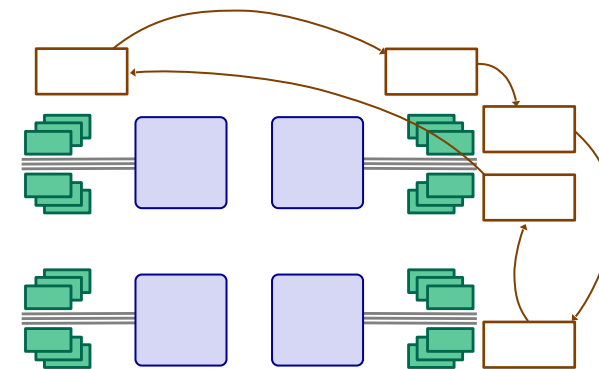
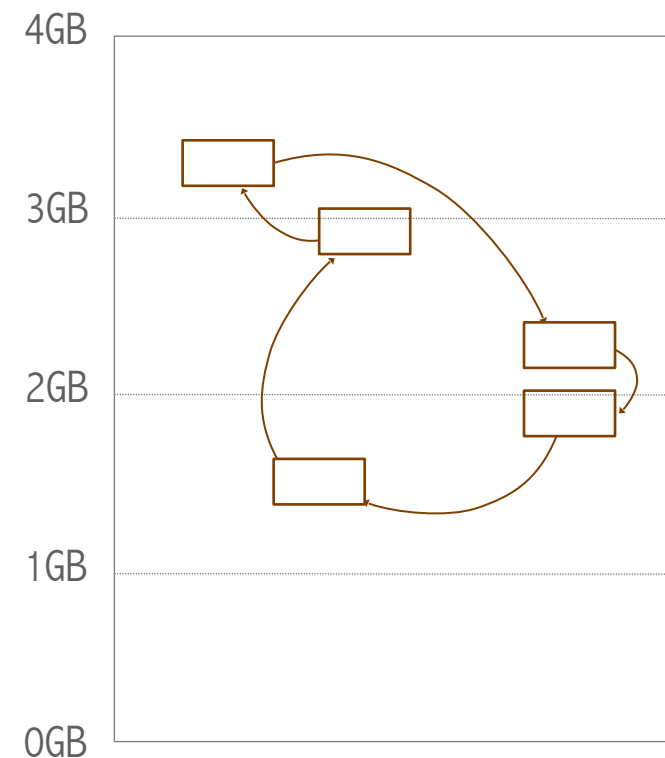
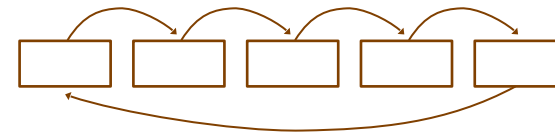


Logical vs. Real Data Structures

The scatter is unimportant as long as Ln and TLB caches hide all latencies. Otherwise **you must explicitly arrange for** a better memory ordering.

There is no silver bullet to make this problem go away.

Custom **application-aware** memory managers, such as pool / slab / arena allocators, other **data structure changes**, and **affinity hints** are the tools.



About Shared Memory

Shared memory is not special – it is completely natural and widely used on modern systems, with many ways to initiate sharing:

Calling **mmap()** on a file in multiple processes can be used to create shared read-only or read-write mappings, on any file region. Example: shared library **position independent code**. One way to share static read-only data is to wrap and load it as a shared library. Suitable use of **mmap() + {f,m}advise()** can map windows of the OS buffer cache and provide hints on future use.

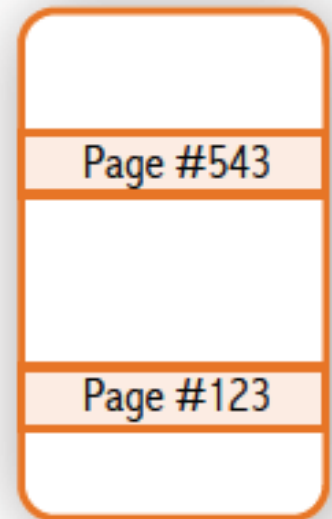
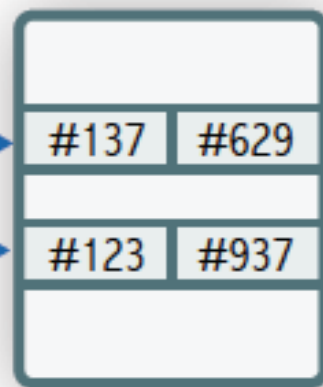
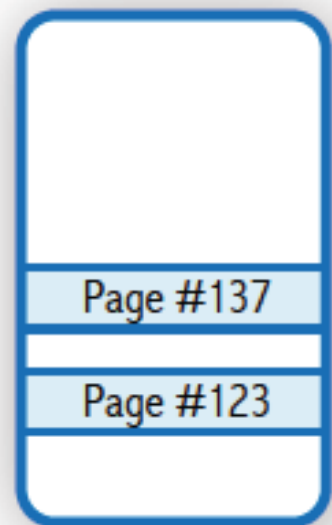
Calling **fork()** without **exec()** makes copy-on-write shared memory of the entire process address space; writing to a page after **fork()** creates a private copy. One of the simplest ways to create **writable transient shared memory** without file association is to use anonymous **mmap()** and then call **fork()**.

It's also possible to create **persistent named shared memory** with **shmget()**.

Pages can be **shuffled around** with **vmsplice()**, **tee()** and **remap_file_pages()**.

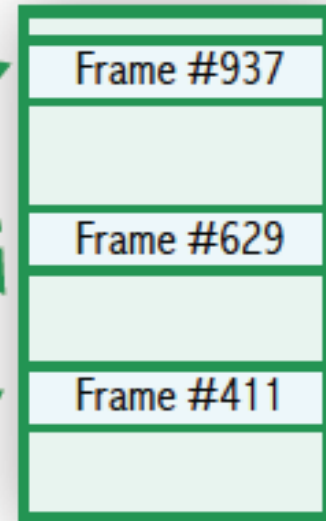
About Shared Memory

Process B Virtual Address Space

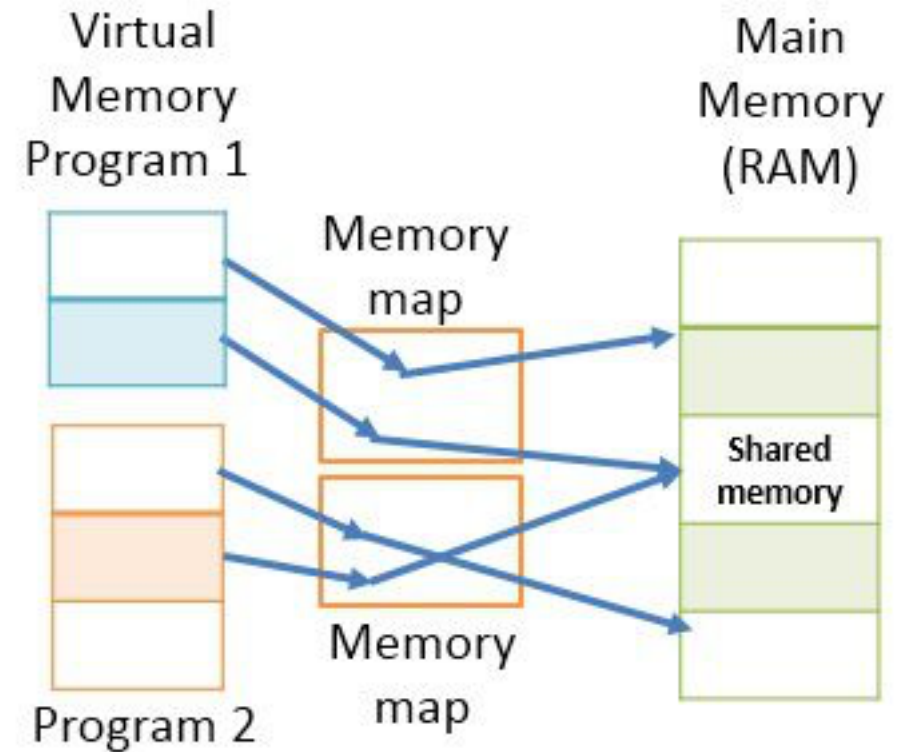


OS Kernel Page Tables

Process A Virtual Address Space



Physical Memory

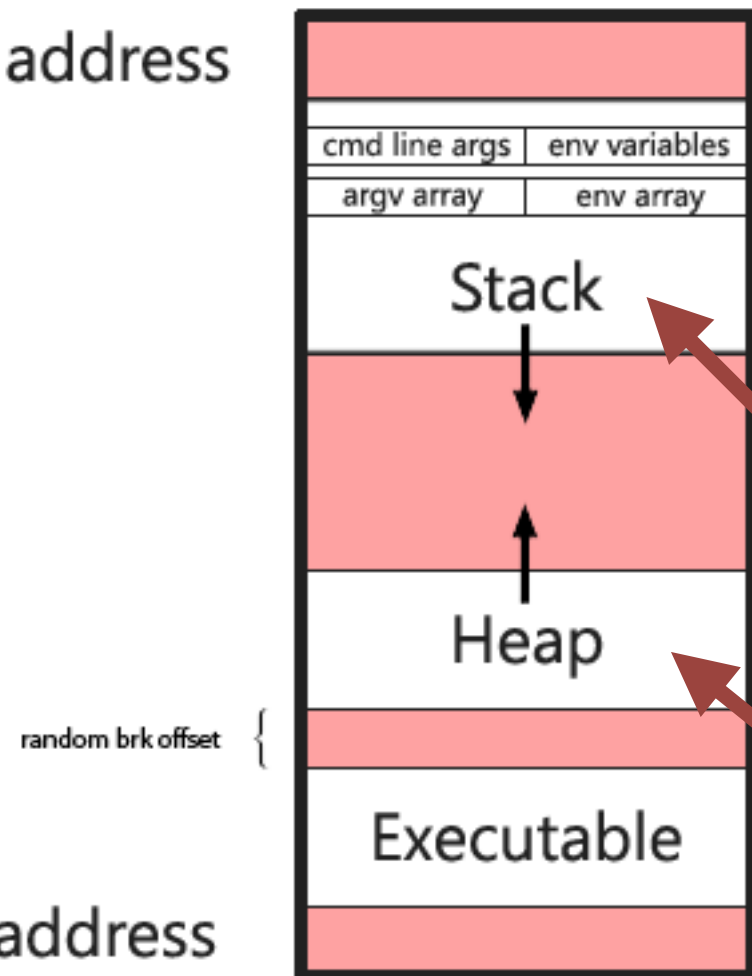


B's page #137 and A's page #123 are mapped to the same physical frame #629, creating shared memory.

#629 could be a read-only page of common library code, writeable memory created with `mmap() + fork()` or `shmget()`.

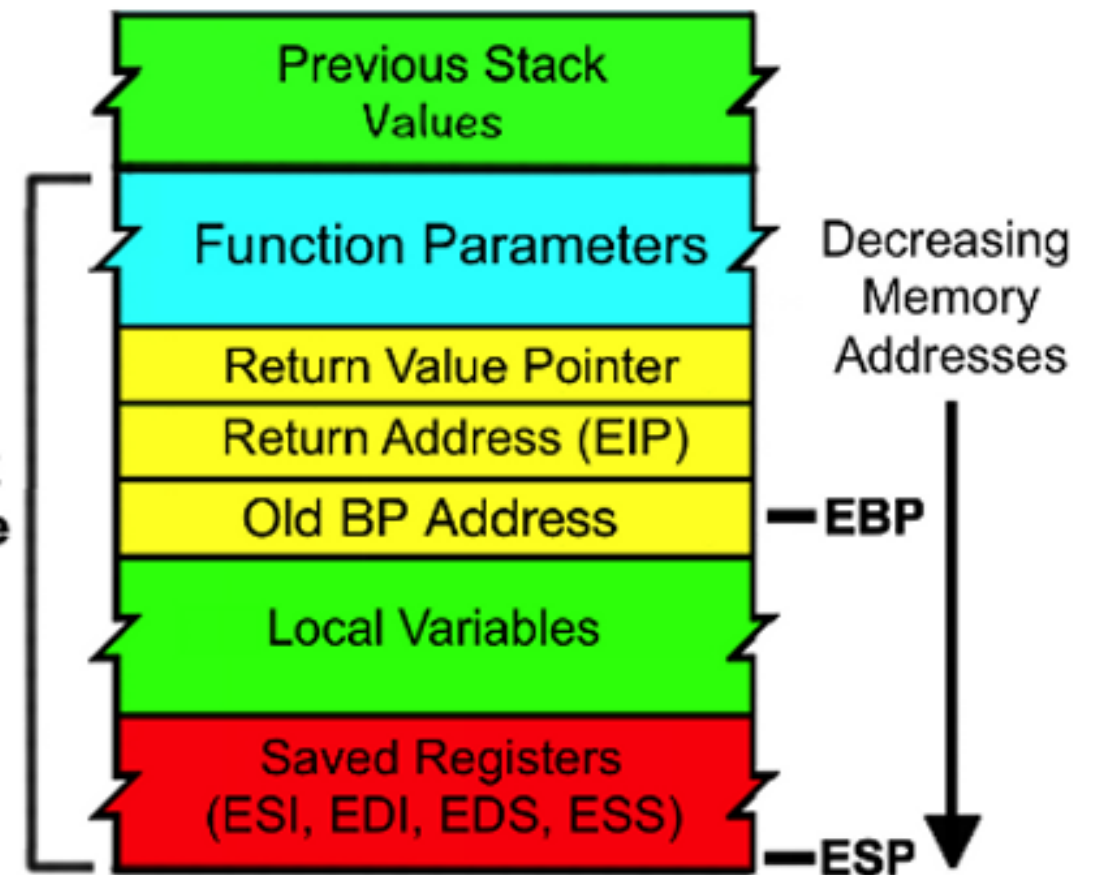
THE VIRTUAL MEMORY

High address



Low address

**Stack
Frame**



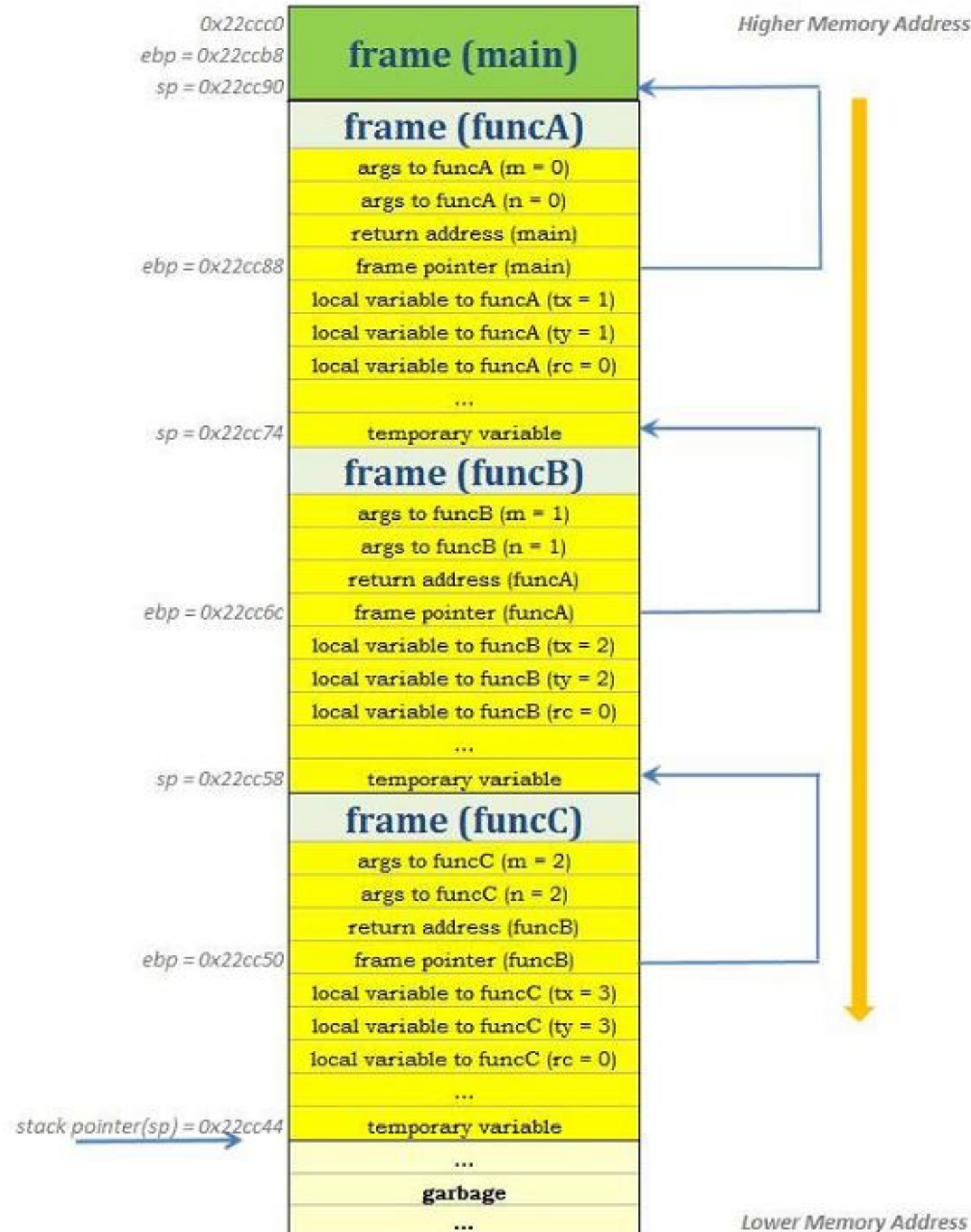
```
void func(int n) {
```

```
    int k[n];
```

```
    auto * p = new int[n];
```

```
}
```


Stack frames a.k.a Activation records



```
Perry: ~/stack_frames.c
1 int funcA(int, int);
2 int funcB(int, int);
3 int funcC(int, int);
4
5 int main() {
6     int tx = 0;
7     int ty = 0;
8     int rc = funcA(tx, ty);
9     return rc;
10 }
11
12 int funcC(int m, int n) {
13     int tx = 3;
14     int ty = 3;
15     int rc = 0;
16     rc = tx + ty;
17     return rc;
18 }
19
20 int funcB(int m, int n) {
21     int tx = 2;
22     int ty = 2;
23     int rc = 0;
24     funcC(tx, ty);
25     rc = tx + ty;
26     return rc;
27 }
28
29 int funcA(int m, int n) {
30     int tx = 1;
31     int ty = 1;
32     int rc = 0;
33     funcB(tx, ty);
34     rc = tx + ty;
35     return rc;
36 }
```


memory (Heap) management Runtime

C++ / std
new, delete
aligned_storage
std::allocator
make_unique
make_shared
vector, list
map,
unordered_map

Posix Libs

malloc, calloc
realloc
free
posix_memalign
aligned_alloc

KERNEL/OS

brk, sbrk
mmap, munmap
madvise

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- If your results are incorrect, buggy, or unreliable, none of the rest matters.*

#2: Memory overhead, alignment & churn matter.

- Badly coded good algorithm \approx bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.*

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.*
- Virtual address locality – address translation capacity is limited.*
- Kernel memory locality – share memory across processes.*
- Physical memory locality – non-uniform memory access issues.*

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- *If your results are incorrect, buggy, or unreliable, none of the rest matters.*

#2: Memory overhead, alignment & churn matter

- *Badly coded good algorithm \approx bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.*

#3: Locality matters, courtesy of the memory wall.

- *Cache locality – stay on the fast hardware, away from the memory wall.*
- *Virtual address locality – address translation capacity is limited.*
- *Kernel memory locality – share memory across processes.*
- *Physical memory locality – non-uniform memory access issues.*

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- If your results are incorrect, buggy, or unreliable, none of the rest matters.*

#2: Memory overhead, alignment & churn matter.

- Badly coded good algorithm \approx bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.*

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.*
- Virtual address locality – address translation capacity is limited.*
- Kernel memory locality – share memory across processes.*
- Physical memory locality – non-uniform memory access issues.*

Memory Overheads

- **Virtual Memory**
 - **Size (VSZ): not a real issue**
 - **Fragmentation: can become a real issue in particular for long running jobs**
 - **reboot machine time to time?**
- **Resident memory**
 - **Size (RSS): IS an issue: swapping is not an option**
 - **Churn: is an issue in particular if triggers system-calls**
 - **cpu overhead, fragmentation**

Memory Monitoring: @System level

- `/proc/meminfo` : stat at node level
 - `cat /proc/meminfo | grep -i anon`
- `ps (top)`: stat at process level
 - `ps -eo pid,command,rss,vsz | grep a.out`
- `/proc/[pid]/smaps`: details at process level
 - `pmap -X yourpid (| tail -n 1)`
 - parse it with a small C++/python program...
- `strace` : real-time or summary for system calls
 - `strace (-c/C) —e trace=memory ./a.out`

Memory Monitoring: @malloc level

- for jemalloc *mallctl* function provides a general interface for introspecting the memory allocator
 - <http://jemalloc.net/jemalloc.3.html>
- see [memory_usage.cc](#) for a simple, robust wrapper
- cpu overhead
 - `std::chrono`
 - perf record/report

Key Memory Management Factors

Many factors at different levels: physical hardware, operating system, in-process run-time, language run-time, and application level.

#1: Correctness matters.

- If your results are incorrect, buggy, or unreliable, none of the rest matters.*

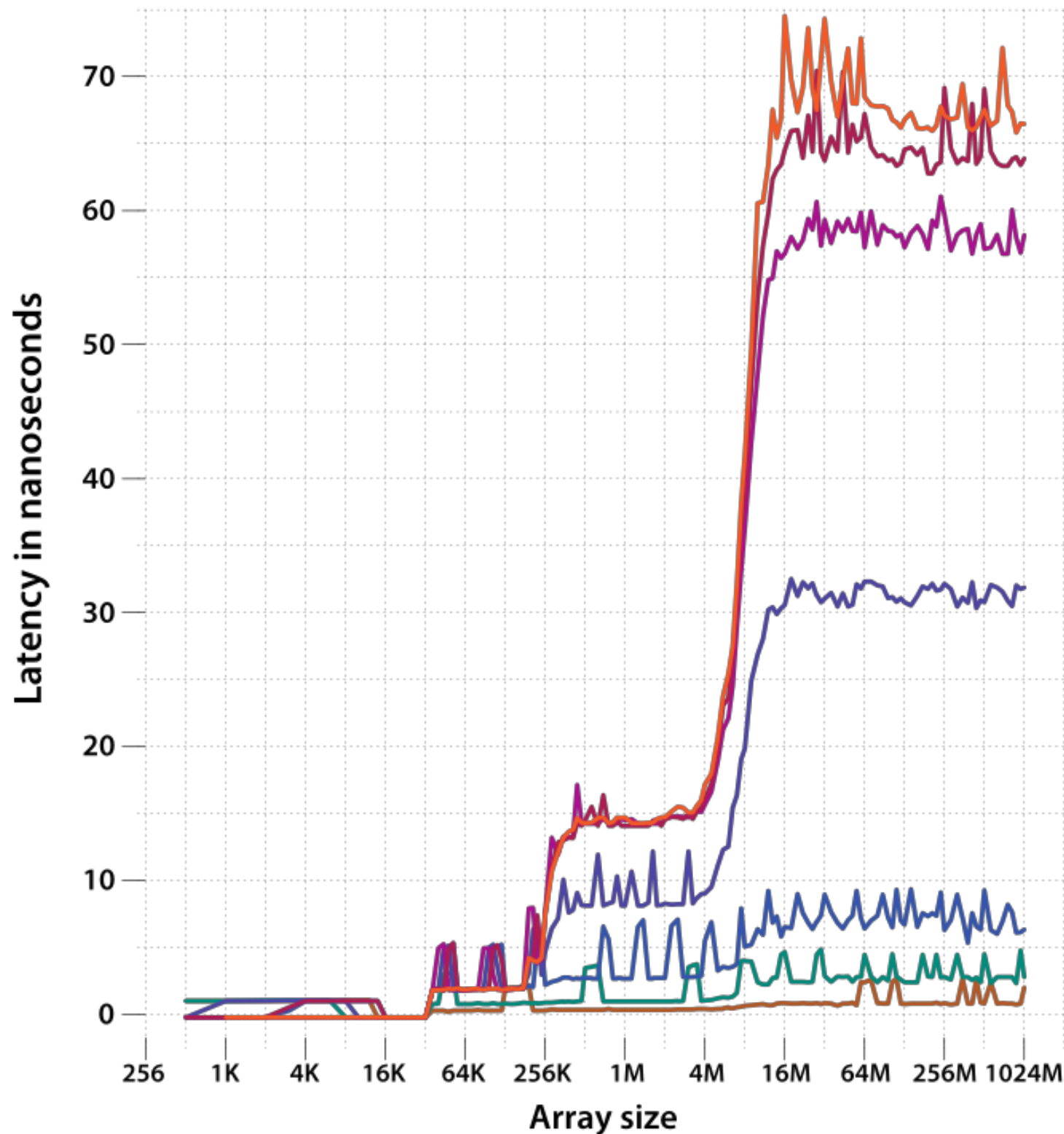
#2: Memory overhead, alignment & churn matter

- Badly coded good algorithm \approx bad algorithm. If you spend all the time in the memory allocator, your algorithms may not matter at all.*

#3: Locality matters, courtesy of the memory wall.

- Cache locality – stay on the fast hardware, away from the memory wall.*
- Virtual address locality – address translation capacity is limited.*
- Kernel memory locality – share memory across processes.*
- Physical memory locality – non-uniform memory access issues.*

Memory latency, Linux 2.6.28 x86-64
Intel i7 940 2.93 GHz, 6GB



[LMBENCH 2.5 results for array strides 16, 32, 64, 256, 512, 1024B]

The Memory Wall

Average memory access time
 $= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty.}$

I/D\$: L1 hit = 2-3 clock cycles.

I/D\$: L1 miss, L2 hit = ~ 10-15 cycles.

TLB: L1 miss, L2 hit = ~ 8-10 cycles.

TLB: L1 miss, L2 miss = ~ 30+ cycles.

What happens when you drop to memory?

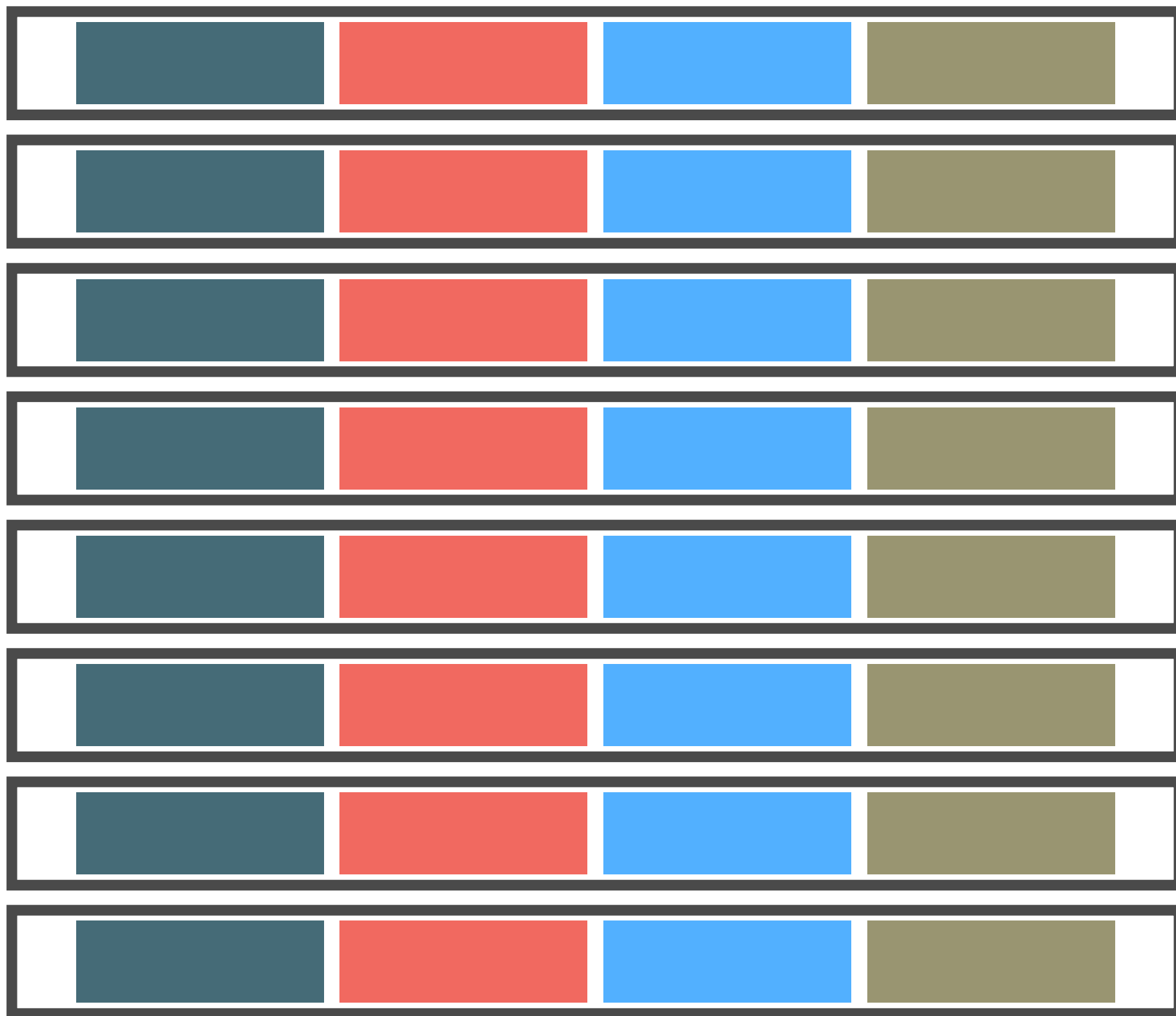
Intel Netburst Xeon (Pentium-era) memory latency was 400-700 clock cycles depending on access pattern and architecture.

AMD Opteron, Intel Core 2 and later CPU memory latency is ~200 cycles (times any NUMA overhead if crossing interconnect).

Good cache efficiency matters.

Array of Structures

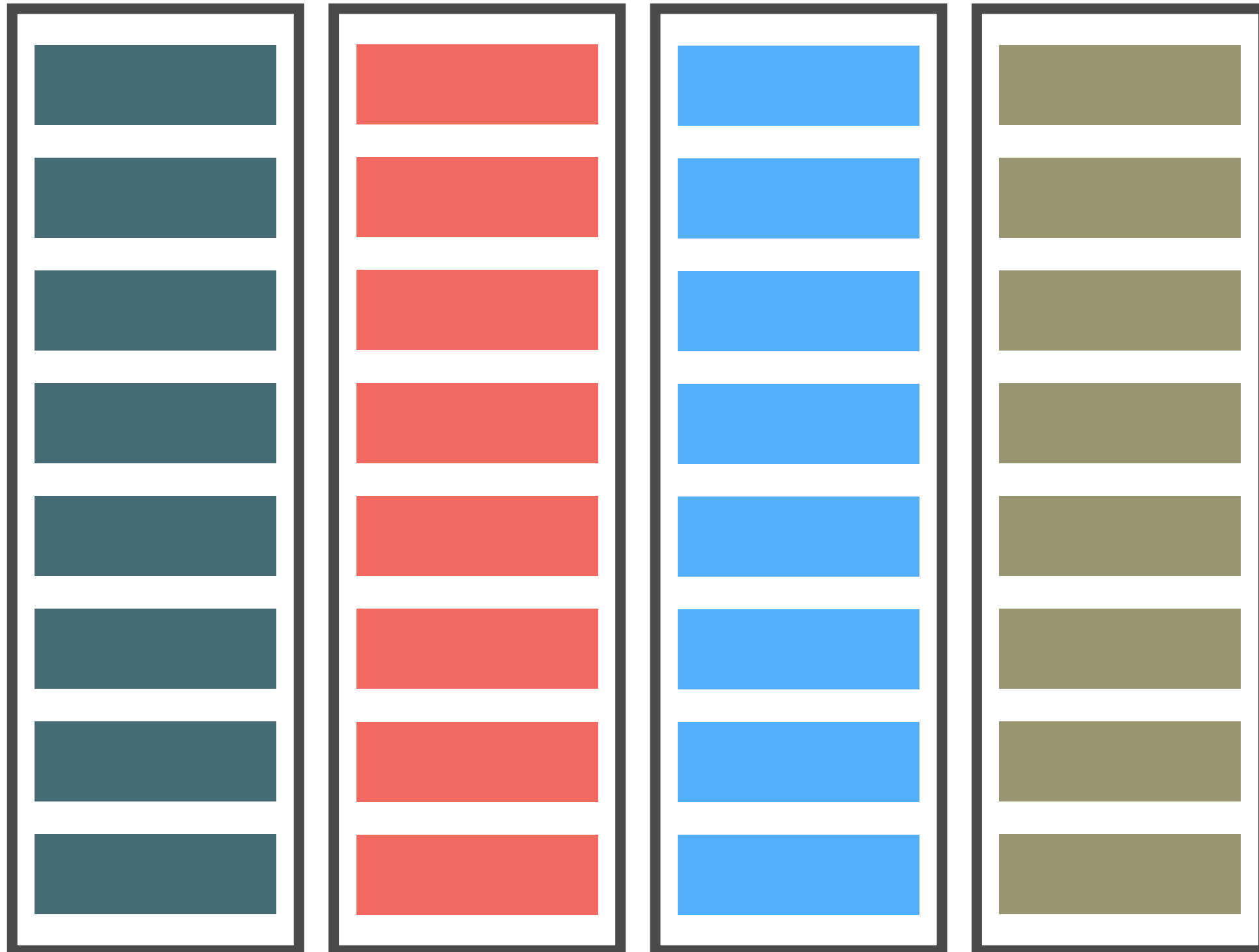
T matrix[N][4];



std::vector<  >

Structure of Array

T matrix[4][N];



```
struct {std::vector<■>; std::vector<■>; std::vector<■>; std::vector<■>;}
```

Matrix and Vectors

```
constexpr uint32_t M=6, N=10000;
#ifdef USE_CPP
std::cout << "using c++ vector and arrays" << std::endl;
std::array<std::vector<float>,M> a{std::vector<float>(N),std::vector<float>(N),std::vector<float>(N),
                                std::vector<float>(N),std::vector<float>(N),std::vector<float>(N)};
std::vector<std::array<float,M>> b(N);

std::array<float,M> v;
std::vector<float> w(N);

#else
std::cout << "using C arrays" << std::endl;
float a[M][N];
float b[N][M];

float v[M];
float w[N];
#endif
std::cout << "a["<<M<<"]["<<N<<"] " << &a[0][0] - &a[0][1] << ' ' << &a[0][0] - &a[1][0] << std::endl;
std::cout << "b["<<N<<"]["<<M<<"] " << &b[0][0] - &b[0][1] << ' ' << &b[0][0] - &b[1][0] << std::endl;
```

Iterating...

```
for (uint32_t j=0; j<N; ++j)
  for (uint32_t i=0; i<M; ++i) a[i][j] *= v[i];
```

```
for (uint32_t i=0; i<M; ++i)
  for (uint32_t j=0; j<N; ++j) a[i][j] *= v[i];
```

$a[i][j] \Rightarrow b[j][i]$ $v[i] \Rightarrow w[j]$

8 variants in total...

let's time them!

(mind compiler's smart optimization)

Wrapping Up

The CPU – memory performance difference has profound impact.

*Operating systems create illusion of one flat virtual address space. In reality the **virtual memory is divided into pages**, and **pages are mapped to physical memory**. Performance critical application must account for this in their design for both data and code management.*

*A process \approx file-backed page mappings for code and read-only data plus anonymous page mappings for stack, heap and global data. Creating many memory regions, for example by loading many shared libraries, harms performance because **good performance requires static page working set which fits in TLB**. Frequent page table changes are costly, some operations require a system-wide stall to synchronise the memory views of all the processors.*

Shared memory is created by pointing pages tables of several processes to the same physical memory pages. Shared memory is common place, and there are numerous convenient ways to create sharing.

Exotic Efficiency Issues

Applications may need to become NUMA aware.

May have to if on NUMA hardware, and either make significant use of concurrency and shared memory (multi-threading or multi-processing); or need more memory than a single physical node has. Read up on numactl.

Poor cache use, not getting enough out of prefetching hardware.

Make sure you use SoA/AoS data structures, then see the other sessions this week on cache awareness, proper strides, alignment, collision avoidance, SIMD, and which tools to use identify problems and possible solutions.

Multi-threaded systems may suffer from cache line contention for heavily accessed data (e.g. locks). Lots of research out there; typical solution is finer grained locks, or eliminating locking using e.g. read-copy-update (RCU). Use multithread aware allocators (like [jemalloc](#) , [TCmalloc](#)).

Killed by large page tables or TLBs? Look into using huge pages.

Summary

Memory management is expensive

Real-world limitations of CPUs and programming languages make memory management a significant factor in overall performance. The solution will vary with technical evolution. If you missed everything else, remember this: get the latency down. May mean you have to design to use hardware-aware AoS/SoA data structures.

No silver bullet

There's no silver bullet for making your applications scream. For top performance you have to invest in real understanding and custom application-specific solutions. Beware memory churn in particular.

Know your tools

There are tools out there which will reduce the mysteries a lot. Now we will combine several of them for more serious exercises!

For the child nerd in all of us...



*Old "arcade" games did not have enough raw CPU power to copy memory around, nor enough memory to store whole levels as big images. They relied on the ability of the (graphics) hardware to "compose" scan-lines from predefined **tiles**, superimposing the result with **sprites**(e.g. the player) images. Tiles and sprites were actually sitting at fixed locations.*

For the teenage nerd inside all of us...



<https://www.youtube.com/watch?v=mxfmxi-boyo>

The video is generated (in realtime) with a 177KB
executable on 2007 hardware